;login:

THE UNIX NEWSLETTER

Volume 6 Number 3 March 1981

CONTENTS

Guidelines for Submission of Newsletter Material

I would like to use the modern text preparation and communications facilities of UNIX to as great an extent as feasible in the preparation of the Newsletter. I have established an account on our PWB/UNIX system so that those who can provide us with machine manageable material can do so. The telephone number is (512) 474-5511. The login name is login and the password is usenix. (The system is also host utexas on the ARPANET.)

For those submitting paper copy of material, please produce your copy on a daisy wheel printer or similar high quality printing device. Line printer produced copy is typically not adequate for reproduction. Copy should be on 8 1/2" by 11" paper with a 1" margin on left, right, and bottom and 1 1/2" margin on top.

U. S. Mail submissions should be addressed to:

Login Newsletter
Computation Center
The University of Texas
Austin, Tx 78712

Attn: Wally Wedel

Uni-ops Association Formed

A new users group is being formed which is designed to facilitate communications between users of UNIX and UNIX-like systems. The group plans to issue a monthly newsletter called "Pipes and Filters". A National convention is scheduled for October 1981 in San Francisco.

Membership in Uni-ops is open to any interested person at $24 a year. Uni-ops mail address is Post Office Box 5182, Walnut Creek, California 94596. Telephone is (415) 933-9564, mornings.

News about UNIX

BIZCOMP Series 1030 Intelligent Modem

Samuel J. Leffler
Sytek, Incorporated
1153 Bordeaux Drive
Sunnyvale, California 94086
(408) 734-9000

With the release of Version 7 UNIX, many people started to use the UNIX to UNIX copy (uucp) utilities distributed with it. However, for most sites, uucp use has been severely limited by the availability of an Auto Call Unit (ACU) with which to initiate transactions. This isolation has become even more keenly felt now that a UNIX User's Network has come into existence and grown to a fairly large size. This note describes our experiences with the BIZCOMP Series 1030 Intelligent Modem. The BIZCOMP is an inexpensive ACU which we have recently gotten, and are using with uucp to become an active host on the uucp-based UNIX network.

A 1030 is actually a 300 baud, Bell 103, modem with a microprocessor of some sort inside (I haven't gotten a chance yet to open it up to see what's inside). The modem may be used in auto-answer mode or, by simple ASCII keystrokes, can be configured to originate calls. When acting as an auto-dialer the BIZCOMP is supplied ASCII commands to dial a given phone number, then places the call, and if successful goes into a ``transparent mode''. To break a connection one sends the BIZCOMP a programmable escape sequence. The BIZCOMP also comes in several factory options: a -068 options silences echoing of the commands supplied it, -08 options suppresses the transmission of an ``auto-id'' character upon completing a connection (used by two cooperating BIZCOMPS to synch up). Our BIZCOMP cost less than $500. A comparable DN-11 like ACU requires two machine ``ports'' on a system to establish a connection and costs upwards of $2000 (I believe for VADIC ACU's). A future model, the 1022, is supposed to cost $595, is more oriented for pure ACU use, but won't be available till March at the earliest.

Given this new toy I couldn't resist integrating it with uucp and another program, tip, which is similar to the Version 7 utility cu (used to provide a ``virtual terminal interface'' to another UNIX system). The job of talking to the BIZCOMP was tedious at first, but eventually handled for tip. The general procedure is:

1. try to synch up the modem by flushing input queues of the associated terminal, then set its prompt to something well known (I used a ``>'' but discovered that this echoed as ``>\0'').

2. define the sequence of characters which will break a connection and hang up the phone (I use ^Q^U^I^T for reasons which will be described below)

3. just to be safe kill the auto-answer on the modem

4. set the type of dialing to be performed to tone dialing or pulse dialing (tip and uucp data bases are consulted to decide which is best for the destination host, but see below)

5. dial the host with a repeat dial command -- this causes the BIZCOMP to continuously dial the phone number till it establishes a connection, or until any character is sent to the BIZCOMP to abort the dial

6. if a ``NO CONNECTION'' message comes back you got a busy signal, otherwise you should get a ``CONNECTION'' message

7. carry on your dialogue with the destination host

8. break the link with the escape sequence defined above

This procedure should be tempered with a couple of notes. When I initiate dialing I also set up a timeout of 60 seconds. The repeat dial command goes on forever, so this must be done. I had previously tried a one shot dial, but found that this was very unreliable, so resorted to the brute force way.

The question of using tone dialing or pulse dialing is pretty much a mystery to me. It is based on what kind of local exchange you have (certain exchanges don't support tone dialing). Tone dialing is the preferred method used by the phone company, but often was found not to work (even to local numbers, though we have a local exchange which should support it). Therefore, I tried dialing numbers by hand and adjusted system tables to reflect my findings. The majority of the phone numbers I use are dialed with pulse dialing.

The use of the ``^Q^U^I^T'' escape sequence turned out to be a stroke of luck when used with the tip program. Since ^Q is normally the X-ON character used by UNIX when a terminal is in TANDEM mode, it is not possible for a user to intentionally break a phone connection when using tip. This is important to us because tip is outfitted to maintain accounting of ACU usage (phone number, length of time on the line, etc.).

The addition of the BIZCOMP to uucp was also quite simple. The major problem was that uucp doesn't think you're going to use anything but a DN-11 like ACU. Consequently a bit of hacking was performed on one routine in order to add the required code.

While we haven't been using the BIZCOMP for very long, it seems to be working fine. For the impoverished UNIX site it appears to be a nice inexpensive addition. Any and all software I've developed for the BIZCOMP is available from me for free.

MX - An indirect driver for multiplexing virtual ``tty'' lines

Piers Lauder
Basser Department of Computer Science
Sydney University

The "*mx*" driver on Unix implements virtual "*tty*" lines, multiplexing them onto *physical* "*tty*" lines via a simple *protocol*. The protocol introduces a traffic overhead *of* approximately 20%, and includes flow control. "Mx" is most often employed in inter-machine networks where the number of physical connections is limited, or is used by concentrators for terminals and line printers.

Introduction

The *mx* driver has spread around the Sydney Unix Network as an easy way of getting *many* virtual connections out of a few expensive long distance connections. The *mx* protocol is essentially a communications protocol tailored for ease of grafting onto the Unix *tty* driver. It is also used to interface different networks to the Unix *net*, *and* to connect slow terminals via concentrators onto higher speed lines. From the point of view of a user, an *mx* port behaves exactly as a standard Unix *tty* interface.

Protocol

The *mx* protocol multiplexes "*mx ports*" onto "*real lines*" and essentially just separates data for each port *on a line*. The *protocol* blocks data with a 4 byte header:-

```
start-of-header;
port-id;
data-byte-count;
inverse-byte-count;
data ...
```

This four byte header is easily recognized and checked for consistency with the repeated byte count. The data length is kept small, in the region of 40 bytes. Flow control is implemented by empty blocks with the first byte count zero, and the second byte count representing the flow rate granted before receipt of the next flow control block. It is important to note that flow rates granted are not cumulative, and thus hung lines can be restarted via a simple timeout mechanism.

Interface

The tty interfaces provided by the mx driver appear to the user to be identical to ordinary tty lines as implemented by other Unix tty drivers. In fact the user interface is provided by the common routines implemented in the file tty.c. The device interface uses device driver routines via the cdevsw procedure array and assumes that they manipulate common Unix tty structures. A few changes have been made to the tty.c routines to identify those lines that have indirect drivers such as mx hanging off them. This is used to vector input through a pointer in the tty structure to the input handling routine in the indirect driver. A timeout scan in the mx driver is used to keep up output flow by examining the virtual and real output queues.

Output

When a write is made on an mx line the data is accumulated on the mx output queue until either the maximum block size is reached, or the user write call terminates. At this point output flow rates are checked and used to limit the next stage. Then the data is blocked up with its protocol header and transferred to the real driver's output queue, for later transmission.

Input

Input bytes via the real device are vectored through ttyinput which in turn calls mxinput. Mxinput examines the bytes for protocol headers using a simple state machine. When a flow control block is identified the "flow enabled" count is reset. When data blocks are identified, the bytes are passed via ttyinput to the appropriate "mx tty" input queue for retrieval by a user read call. A flow control grant block is generated whenever the user input queue is low enough. This is checked as soon as the byte count is encountered, thus providing a certain amount of "read-ahead".

Inefficiencies

For maximum efficiency, user writes should be of the order of the maximum block size. However many Unix programs generate 1 byte writes to tty interfaces, thus causing protocol overheads of 400%, and most line degradation is caused this way. Flow control is granted in small lots equivalent to the maximum size of a data block, doubling the effective overhead. The recursive calls of the tty routines at interrupt time cause a much greater system overhead in indirect drivers implemented this way.

Conclusions

As the real traffic rates on high speed lines connected to terminals are much lower than the maximum, the protocol overhead of the mx driver is acceptable. Furthermore, output and input data rates for high speed terminals typically differ by a factor of 10 to 1, so two ports operating in opposite directions on the same line have a low impact on each others apparent speed.

Implementation

All the mx driver needs to know about each real tty line is the device id (major, minor), the number of ports to be multiplexed onto the line, and the baud rate.

These details are initialized in the configuration structure "mxdev" - just alter it to reflect the local conditions. A tty structure is declared by "mxtty" for each port, so, on some small systems with kernel memory limitations, the number of ports should be kept small.

Mxtty is allocated as a contiguous array, and therefore minor device numbers for the special files must be contiguous. This means that if you want to change the number of ports on the first line, the special files will have to be remade for all the other lines.

The structure "mxmap" maps each real tty onto the "ported" tty structures, and contains current information about block transfers.

SUN - The Sydney Unix Net

Piers Lauder
Basser Department of Computer Science
Sydney University

ABSTRACT

The Sydney Unix Net is a simple implementation of a user initiated file transfer facility. SUN is a "host supported" network, and considerations of low overhead have lead to an efficient design. The network is self configuring with an optimal routing algorithm.

Introduction

SUN consists of linked hosts (nodes) with unique names, where links between nodes may be unreliable and quite slow. Using this network, a simple system has been developed to provide reliable host-host file transfer. The system is implemented in two levels. Level one provides an error free path between nodes, and level two implements a host-host protocol, and maintains a network topology file for routing calculations. Files are transmitted through the network until they reach their destination where they are spooled for later collection by the remote user, who is informed by mail of the arrival of files from the network.

User Interface

The user interface is as simple as possible. A network address is defined as a _username_:._I_ _host_ pair, i.e. the name of the user on the remote host is separated from the remote host name by a _colon_. This form of address is understood by the mail programs, some print commands, and the _netsend_ command.

Two special file transfer types are recognized in addition to user-to-user file transfers, namely _mail_ and _print_. Mail is automatically delivered on the remote host by invoking _mail_, and _print_ files are handled by invoking the print spooler, but user files are spooled in a holding directory for later collection by the designated user. The arrival of files is notified by mail to the user. To complete the file transfer (and assume ownership of the file), a user must invoke the _netget_ program to retrieve the file from the holding directory. Uncollected files can be easily discarded from time to time.

Design

The network programs operate on two levels with clearly defined interfaces. The top level (implemented by the program _net_) accepts files for transmission, calculates the next node in the path, and spools the file together with a host/host protocol header in a directory naming the next host. The lower level (implemented by the program _netd_) accepts files for transmission to an immediately connected host, negotiates with the lower level on the remote host for file transfer to start, and then uses a node/node protocol to transfer the file reliably. On arrival at the next node, the file is passed to the upper level for any further routing.

The file transfer mechanism is half-duplex at the lower level, and store-and-forward at the higher level.

Node/Node Interface

The link between hosts provided by the operating system must be a full duplex, byte oriented special file. The name of this special file is that of the node to which it is connected at the remote end. This file is opened for reading and writing by the network daemon responsible for communicating with the next node.

Actual links between hosts may be physical RS232 type lines, either directly connected, or via telecommunications modems, and be handled by the standard Unix _tty_ interface. Many hosts multiplex the physical link using the _mx_ tty interface (mentioned elsewhere) to which the network requires only one port.

Node/Node Protocol

The network daemons communicate with each other over the node-node interface. They use a half-duplex, multi-buffered, positive acknowledgment data transfer protocol. Before file transfer starts, the daemons negotiate the direction of the next transfer. File transfer proceeds with short data blocks enclosed in a protocol _envelope_ consisting of a header and a trailer. The header contains a sequence number used to provide the multi-buffered message flow, and the trailer implements a simple low-cost error detection capability. Messages must be acknowledged (positively or negatively) with a two byte reply consisting of ACK or NAK and the relevant sequence number. Errors cause the re-transmission of all un-acknowledged blocks. Catastrophic error conditions cause a negotiation for file transfer restart.

Daemon/Spooler Interface

The interface between spooler and daemon is defined by queued files. Each daemon maintains a command directory which it scans for command files. Each command file specifies the path names of files for transmission. The spooler chooses the next host for transmission by the fact that the name of the host is also the name of the command directory for the appropriate daemon. On the other hand, files received by the daemon are passed directly to the spooler program.

Host/Host Protocol

The network routing program net (the "spooler" referred to above) prepends a host-host protocol header to each file before spooling it in a daemon directory for transmission to the next host. This header contains the source and destination network addresses together with routing information and file parameters. Each file arriving from the net is examined for this header and re-routed if the destination is not yet reached. Each host through which the file passes adds a host/time record to the routing information in the header. Amongst other things, this information can be used for network performance analysis.

Topology File

The routing information in each file header is used to maintain the topology file. Each host mentioned in the route is connected to the preceding and succeeding hosts, and these links are maintained in the topology file. It is possible for a topology file to become aware of new hosts in this way, however there are special files whose purpose is to maintain network topology files generally. Thus there are "host-up" messages to inform the network of a new hosts, and "host-down" messages to inform the network of broken links. These messages are broadcast around the network by the net programs who also stop any loops. New hosts coming up receive a special file from any immediately connected hosts containing a copy of their topology files, thus immediately informing a new host of the latest network state.

In order to send a file to a remote host, its name must exist in the topology file so that the routing program can find it, unless it is directly connected.

Conclusions

The initial effort producing the software for a usable network took about 2 man-weeks. Since then many requested enhancements have been implemented involving a further 4 man-weeks of work. As it now exists, SUN has proved very helpful in bringing together many people involved in cooperative work in support of our various teaching systems.

Share Scheduling Works!

Piers Lauder
Basser Department of Computer Science
Sydney University

ABSTRACT

The Share Scheduling Algorithm on Unix provides a per-user scheduler on top of the standard per-process scheduler. It acts to provide equitable allocation of the machine between classes of users, and between users of the same class, according to their allocation of "shares" of the machine.

Introduction

At Sydney University, we have implemented a version of the Cambridge Share Scheduling Algorithm as proposed by J. Larmouth of Salford University, and modified by Andrew Hume of AGSM. This scheduler has been very successful in allocating the limited resources of our student teaching machine (overcoming the otherwise unrestricted generosity of Unix) and has also been surprisingly successful as an advertisement for the sophistication of our teaching service amongst people who might otherwise consider Unix to be a toy operating system unsuited for the real world.

Implementation

The Share algorithm allocates a share of the machine to each user based on his history of past usage, his current working rate, and the ratio of his allocated shares to those of other users concurrently using the machine.

The algorithm as implemented was extensively modified by Hume (as described by Hume in his paper "A Share Scheduler for Unix"). This was mainly due to Larmouth's algorithm being batch oriented, whereas an interactive environment requires quicker responses.

The low level scheduler in Unix allocates use of the cpu amongst competing processes based on their priority which decays very quickly. Thus every process competes on an equal basis. It is possible for a user to increase his share of the cpu by running many "asynchronous" processes simultaneously. In order to allocate the cpu between users, regardless of the number of processes, the low level scheduler was changed to use a per-user priority which is added to each process's short term priority. This per-user priority is manipulated by the Share scheduler to affect a user's long term share of the machine.

To calculate the cost of the user's use of the machine, separate charges are made for cpu time, memory occupancy, terminal I/O, backing store I/O, and system services. This cost is added to the accumulating usage, and the accumulating rate. The rate figure decays quite fast, approximately 10% second, whereas the usage decays quite slowly, approximately 10% day. There are thus two separate figures contributing to the calculations for the real share of the machine to be allocated in the next time period, one based on recent working rate, and the other on long term usage. It is therefore possible for high usage users to do a little every now and then. The real share figure is then used to affect the user's overall priority.

The Real Effect

The Share Scheduler has no effect whatever on an under utilized machine, and in practice this means less than 45 users on our VAX 11/780. However, over this, the effects are very satisfactory. To be most useful, there must be, at any one time, two or more classes of user with different priority of access to the resources. For instance, during class hours, students are deemed to have greater rights to a reasonable response than, say, academic staff. This is achieved by an (heuristic) allocation of shares to each class of user based on expected average use of the machine, and bearing in mind the decay rate of the usage. For example, at Basser we allocate 10 shares to first year students (who are expected to use 3 terminal hours per week), 50 shares to academic staff, and 200 shares to staff programmers. This effectively bars academic staff from using the machine during periods of heavy usage as their figure is always fairly high, whereas first year students (whose usage decays substantially between their terminal sessions) can obtain reasonable response even during times of peak loading. On the other hand, the 200 shares allocated to programmers allows them to carry out normal system support programming with reasonable response at all times. All this is in addition to the fair distribution of resources within each class of user, as users who use the machine heavily are not allowed to reduce the response of similarly classed people who are working lightly.

With 45 users experiencing good response, it would appear that each user needs around 2.2% of a VAX 11/780, and it would be reasonable to nominate a share figure, say 0.5%, below which the system should actively discourage users from logging in, although we haven't done this. The consequence of not doing this, is that if a user's share drops below say 0.1% during a session when many more people have logged in, he may get "marooned" with insufficient resources to allow log off (there being no cpu allocation enough to execute the "exit" system call from the shell!).

A user may discover his share during a terminal session, and watch it change as costs are incurred, and he is of course informed of it at login. However the changes can be quite dramatic if for instance a user is the first of 75 to log in after a system reboot. It might be advantageous to inform a user whose share has dropped below a magic figure during a session, so that he has a reasonable chance to log off and let someone else use the terminal to greater advantage.

Tools

There are various system monitoring programs to display the performance of the scheduling algorithm, and nearly all parameters can be changed dynamically. In particular all charges can be varied to reflect administrative policies about system resource costs. At Basser we charge at a fixed rate between 9am and 5pm of a working day, 50% of that until 10pm, and 10% overnight. Also users using the "nice" command get charged less for cpu time as appropriate.

Implementation

The actual scheduler is one page of C code and uses floating point extensively, although Hume has proposed a model using integer arithmetic. On the Vax, floating point is easier as the code is much cleaner, and the floating point instructions are reliable. The C compiler produces extremely inefficient floating point code, nevertheless, running once every 4 seconds, the scheduler consumes less than 0.5% of the cpu time managing 80 users on the VAX. Recoding the algorithm in assembler would reduce this to around 0.2%.

Charges are levied throughout the system at the various "cost" points. This is done by adding a variable charge for the resource type (obtained from a charges array) into a per-user "cost" variable.

The scheduler is invoked by the clock interrupt routine requesting a software interrupt at a low priority, once every 4 seconds.

The algorithm first scans all the per-user structures building up an idea of total rates and shares, and simultaneously decaying the usage and rate figures after adding in the cost. The rate and usage totals are then biased by (tunable) factors reflecting their relative weights in the priority calculation.

A second scan of the per-user structures then calculates the per-user priority to be used by the low-level scheduler. A user's share of the machine is thus reflected in his priority relative to the priorities of all other users.

Example Monitor Output

The following is a typical output produced by one of the monitor programs used to tune the algorithm.

```
Wed Oct 22 16:49:37 1980

Lnodes: 78    Users: 76                 Averages:
Usage factor  Rate factor Total rate     rate      usage  shares
4.224907e-05 1.343931e-03    2172346    21609  8.373e+07 37.1053
Constants:-
syscall · bio    tio    tic click maxnice ufactor rfactor    usagek
   1360  1000    283   8333     1      18       1       2 0.9999834
**
****
****
****
*****
*****
*****
*****
******            *
*******  *        *
******* **        *
*************   ***
0 2 4 6 8 0 2 4 6 8   background = 1
```

There are 76 users logged on, not including the idle process and the system user (root). The "Usage" and "Rate" factors are intermediate values used by the scheduling algorithm, the "Total rate" value includes that of the system and the idle process, whereas the averages are for "real" users only. A real user is one who has been allocated shares. The constants refer to the charges being made for each of the resources, "usagek" is the rate of decay applied to usage figures, effectively 20% per day.

The histogram shows the number of users vs. their priority.

Modifications for Unix on Small CPU's

Kenneth A. Reek
School of Computer Science and Technology
Rochester Institute of Technology
One Lomb Memorial Dr.
Rochester N. Y. 14623

Like most other university environments, we at RIT have needs for computing equipment far greater than can be satisfied by our computing budget, so we make do with what we can get. We are, therefore, another of the few installations running 7th Edition Unix on a "small" PDP-11, that is, one lacking separate instruction and data spaces. We wanted to support at least a half-dozen users running rather large programs, so we decided to solve the size problem and the resulting crunch on buffers by modifying the kernel.

The primary change was to remove the system's buffers from the kernel address space. We had heard that this had been done before, but somehow never got the name of anyone who had done it (or never got around to contacting them). Our change, then, is an independent effort, so we are interested in exchanging ideas with anyone else who has done this.

It turns out that there are only a few places in the kernel outside of I/O drivers where the buffers themselves (as opposed to the buffer headers) are actually accessed. It is these places that are affected. No changes are needed in the I/O drivers, since the device controllers do not know about "address spaces".

Instead of declaring space for the buffers themselves at compile-time (in main.c), we allocate space for them during system initialization by calling "malloc" from "binit". We have only one buffer in the kernel address space which is used as a scratch pad. Our implementation is limited to at most 32 buffers to guarantee that they fall into physical memory whose high-order two bits are both zero, which simplifies accessing them. We have assembly language routines to copy data into and out of the buffers, which is accomplished by "borrowing" the kernel's mapping register number 1, and changing it to point to the desired buffer. That particular register is safe to use because it does not cover the assembly routines themselves (m40.s is less than 8k bytes) and it does not cover any data (the kernel code is larger than 16k bytes). Copying is done at high priority to prevent any interrupt service routines from trying to access the code normally serviced by register 1. After the data is copied into or out of the buffer, the previous values are restored to the map. We consider this to be a very conservative (i.e. kludgy), but safe approach. On an 11/34, it takes about 1.7 milliseconds to copy a buffer, so the only devices that would be affected are those that require interrupt service within that interval, such as a character multiplexor without a silo receiving at 9600 baud. We have nothing like this, so it isn't a problem for us.

The assembly language routines are called whenever the kernel accesses a buffer. Many of the accesses use "bcopy", and in these instances the only change needed is to substitute one routine for the other. For accesses where the code itself pokes around in the buffer (as in updating inodes), we call our routine to copy the desired buffer into the scratch pad, modify it, and then copy it back into the real buffer. Being overly cautious, we protect the scratch pad buffer with code that guarantees its exclusive use by reporting any collisions via a "panic". The only time such a collision occurred was when the system was already crashing for an unrelated reason.

This change frees up the kernel space that is normally taken by buffers so it can be used for other data structures. We now have 25 buffers, and enough space left for lots of inodes, file structures, and the like.

A couple of other changes we have made might be of interest to some users. Rather than using system buffers to hold the super-block of mounted file systems, we allocate space at compile-time for the desired number of mounted file systems. If all the file systems are usually mounted all of the time, this buys a little memory, since the super-block structure is smaller than a buffer and no header is needed. If, however, your system normally has several unused mount entries, then there will be a net loss in space.

We also removed all of the tunable constants from the source files and replaced them with references to a table where the desired parameters are stored. By changing "mkconf", we create this table in c.c, along with all the data structures that depend on tunable parameters. In this way, changing these parameters does not require recompilation of the system (though the symbol table in the C compiler had to be enlarged to handle the additional symbols). Also, by modifying programs such as "ps" and "pstat" to examine the table in the kernel, they need not be recompiled each time the system is tuned. This saves us a lot of time, since we like to fiddle with these things frequently.

We use our PDP-11/34 to support student projects in courses that deal with programming I/O devices and handling interrupts. The single-user systems that we used in the past (of which the 11/34 was one) simply could not handle the increased enrollment in the courses, prompting us to investigate the use of some kind of timesharing. The problem with using a timesharing system for this application is obvious, however. Since many users are working on their programs, users cannot run their programs because doing so would crash the timesharing system.

We currently have 3 LSI-11 microcomputers connected to the system, any of which can be accessed from each of the 7 user terminals. The program that communicates with the LSI's sets things up so that the user's terminal acts like the console terminal for the LSI, allowing the user to make full use of the console emulator and its debugging capabilities. The program also provides users with the capability of sending their programs to the LSI-11 for execution, which solves our timesharing/single-user dilemma. Numerous debugging and development utilities have evolved to further simplify the student's task. With this arrangement, we can support (and give better programming facilities to) a much larger student population than would have been possible by simply buying more single-user systems.

We will send the kernel modifications and any of the other stuff we have to all interested (but authorized) parties who can take 800 or 1600 bpi tapes in "tar" format. Send a copy of the identifying part of your license, along with $10 for postage and handling. Delivery time is an exponential function of the number of ungraded tests on my desk at the moment.

USENIX ASSOCIATION
BOX 8
THE ROCKEFELLER UNIVERSITY
1230 YORK AVENUE
NEW YORK, N.Y. 10021

FIRST CLASS MAIL

FIRST CLASS MAIL